RxJS

Streams in Action

Recap Promise

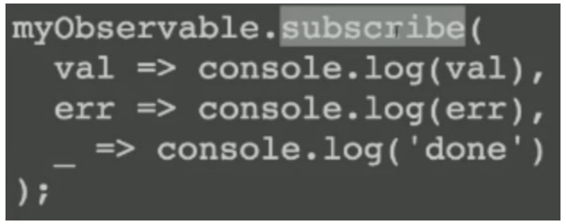
- Asynchrone Callbacks
- Einmal aufgerufen
  - Anschliessend beendet
- Keine mehrfache Verwendung möglich

- `then(onFulfill, onReject)`
- `catch(onReject)`

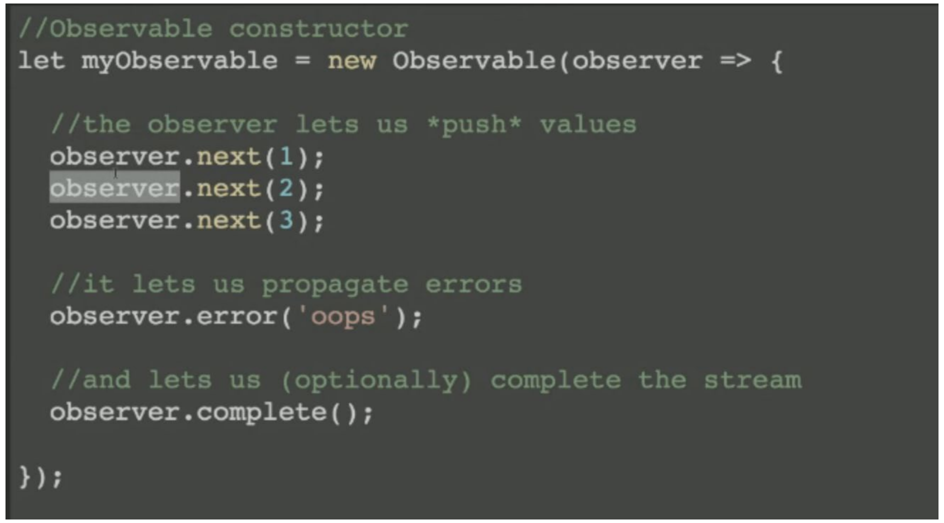
```
doSomethingWithPromise()
.then(receiveResult)
.catch(handleFailure)
```

Was ist RxJS?

- Asynchronous Observable Pattern
  - Programming with ansynchronous data streams
  - Collection that can change over time
  - Observables are like Promises, but work with multiple values
  - Observables can be canceled
  - Map, Filter, Reduce => bring to events



```
myObservable.subscribe(
  val => console.log(val),
  err => console.log(err),
  _ => console.log('done')
);
```



```
//Observable constructor
let myObservable = new Observable(observer => {

  //the observer lets us *push* values
  observer.next(1);
  observer.next(2);
  observer.next(3);

  //it lets us propagate errors
  observer.error('oops');

  //and lets us (optionally) complete the stream
  observer.complete();

});
```

Observable / Observer

- Asynchroner Stream
  - Mit Callbacks
- Beispiele -> Stackblitz

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});

console.log('just before subscribe');
observable.subscribe({
  next(x) { console.log('got value ' + x); },
  error(err) { console.error('something wrong occurred: ' + err); },
  complete() { console.log('done'); }
});
console.log('just after subscribe');
```

Observable

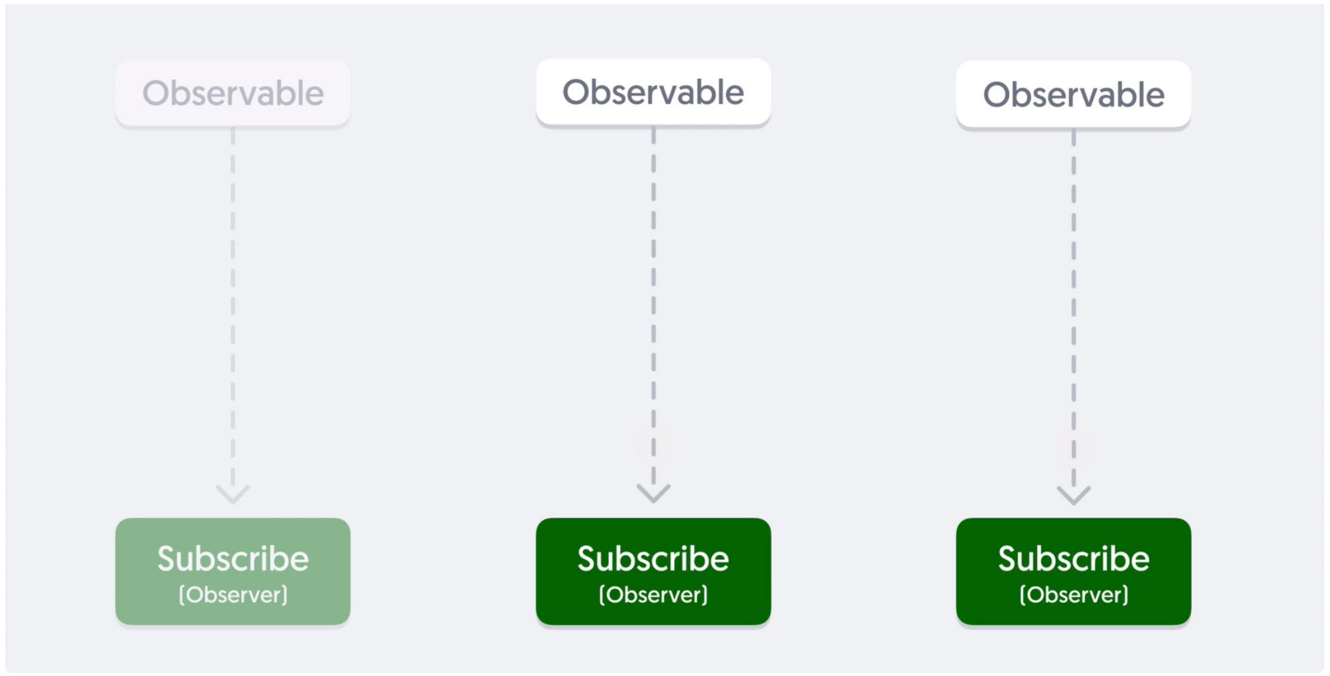
- Single Cast

```
import { Observable } from 'rxjs';

const foo = new Observable(subscriber => {
  console.log('Hello');
  subscriber.next(42);
});

foo.subscribe(x => {
  console.log(x);
});
foo.subscribe(y => {
  console.log(y);
});
```

OUTPUT:

```
"Hello"
42
"Hello"
42
```

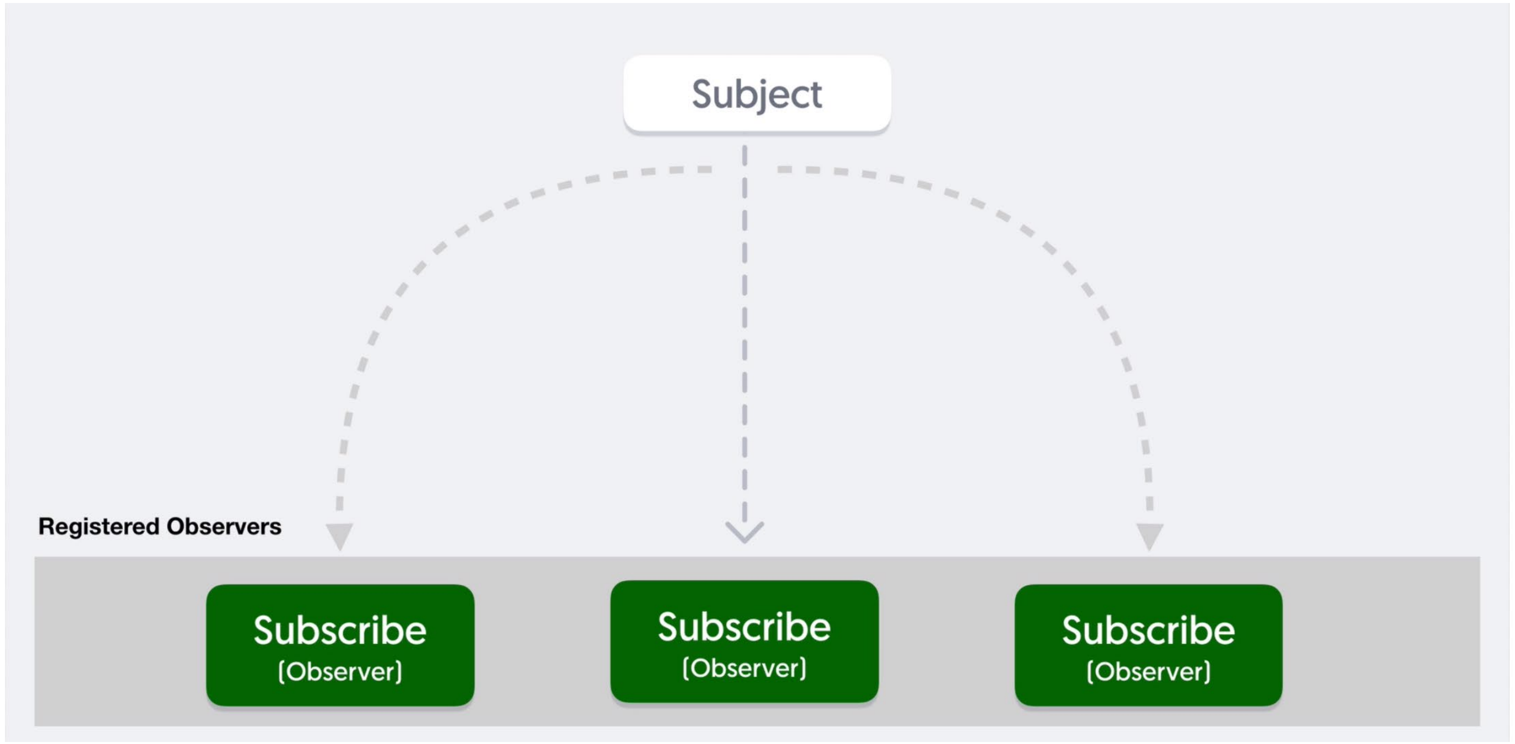
Bild-Quelle: https://ultimatecourses.com/courses/rxjs



Subject

- Multicast
- Beispiele -> Stackblitz

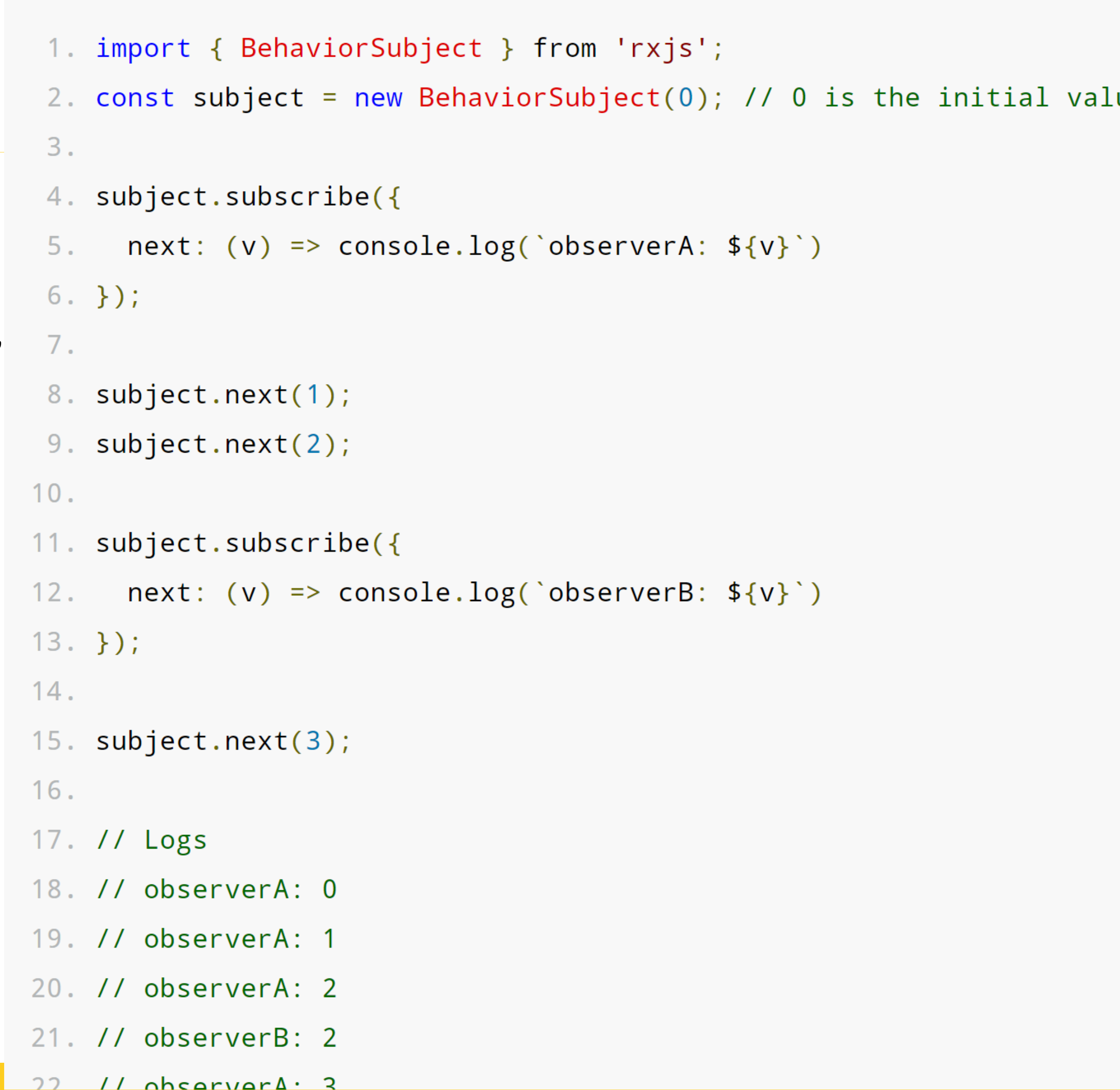
Bild-Quelle:
https://ultimatecourses.com/courses/rxjs



```
import { Subject } from 'rxjs';

const subject = new Subject<number>();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(1);
subject.next(2);

// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

BehaviorSubject

- Gibt bei neuen “Subscriber” immer den aktuellen Wert zurück



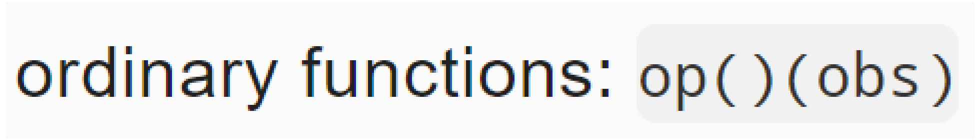
```
import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject(0); // 0 is the initial valu

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(3);

// Logs
// observerA: 0
// observerA: 1
// observerA: 2
// observerB: 2
// observerA: 3
```

Operatoren

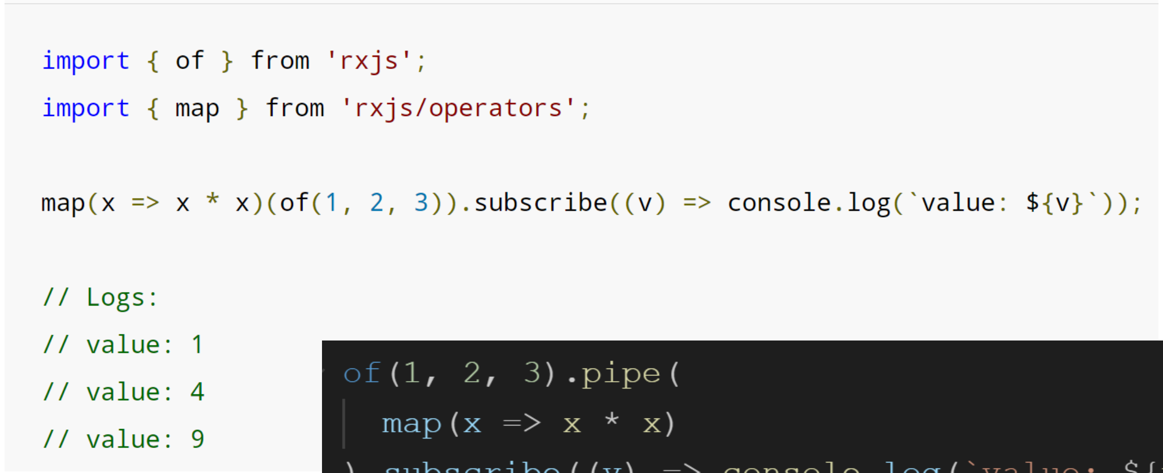
- Die Daten des Streams zu transformieren oder zu bearbeiten
- Pure Function
- Pipable Operators können «gepiped» werden:
  `observableInstance.pipe(operator()) - .filter(…) - .mergeMap(…)`
  - Geben NEUE Observable zurück und modifizieren das aktuelle nicht!
- Wenn du einem Output Observable «subscribst» dann wirst du ebenfalls dem Input Observable «subscribed»
- Creation Operators (e.g. of(1,2,3))

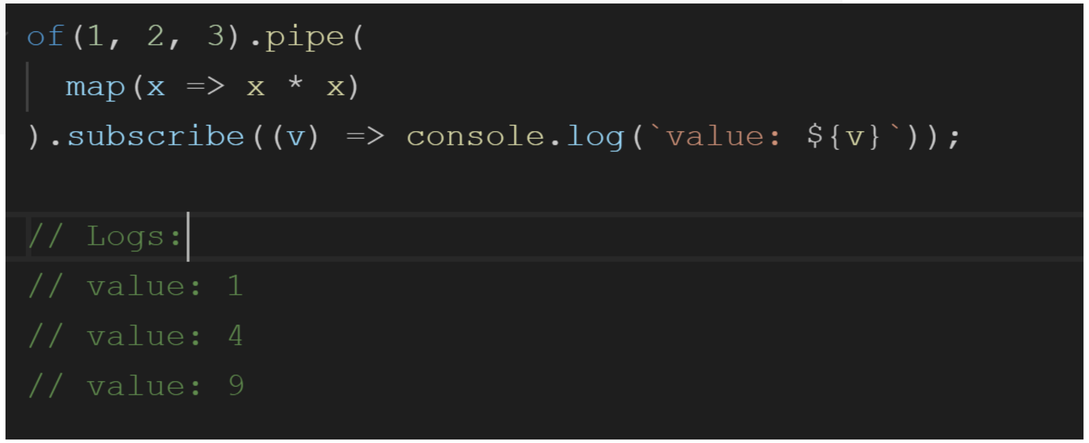
```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(`value: ${v}`));

// Logs:
// value: 1
// value: 4
// value: 9
```

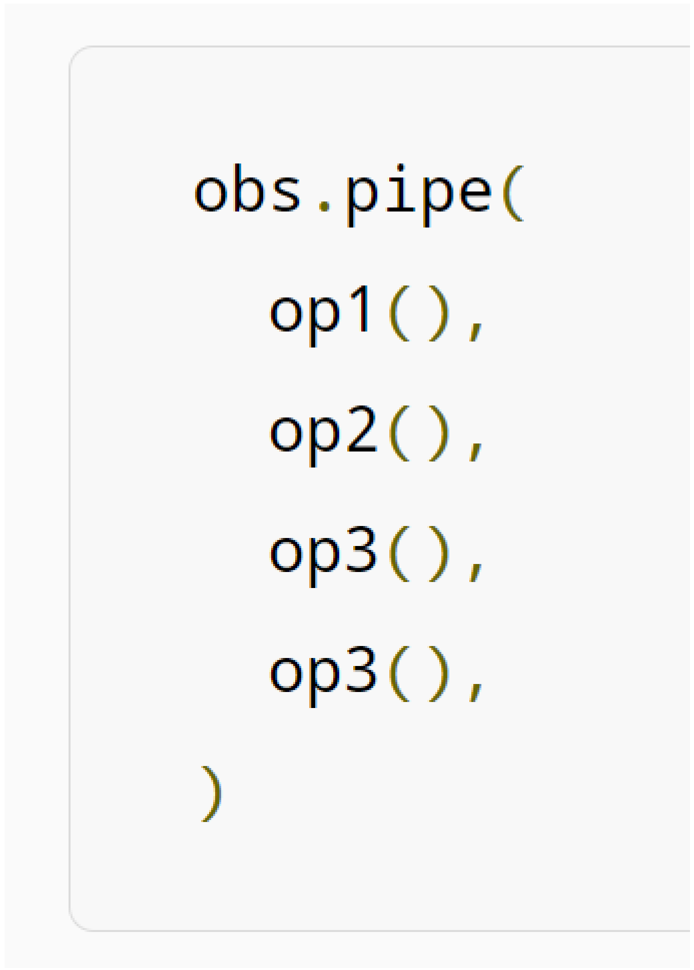
Operators - Piping


ordinary functions: `op()(obs)`

- Function


```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(`value: ${v}`));

// Logs:
// value: 1
// value: 4
// value: 9
```


```
of(1, 2, 3).pipe(
  map(x => x * x)
).subscribe((v) => console.log(`value: ${v}`));

// Logs:
// value: 1
// value: 4
// value: 9
```


```
obs.pipe(
  op1(),
  op2(),
  op3(),
  op3(),
)
```

Operators – Creator

- Erzeugt ein Observable aus..
  - Intervals
  - Werten
  - Events
  - usw.

```
import { of } from 'rxjs';

of(10, 20, 30)
.subscribe(
  next => console.log('next:', next),
  err => console.log('error:', err),
  () => console.log('the end'),
);
// result:
// 'next: 10'
// 'next: 20'
// 'next: 30'
```

```
import { interval } from 'rxjs';

const observable = interval(1000 /* number of milliseconds */);
```

Creator Operators

- `ajax`
- `bindCallback`
- `bindNodeCallback`
- `defer`
- `empty`
- `from`
- `fromEvent`
- `fromEventPattern`
- `generate`
- `interval`
- `of`
- `range`
- `throwError`
- `timer`
- `iif`

Higher order Observable

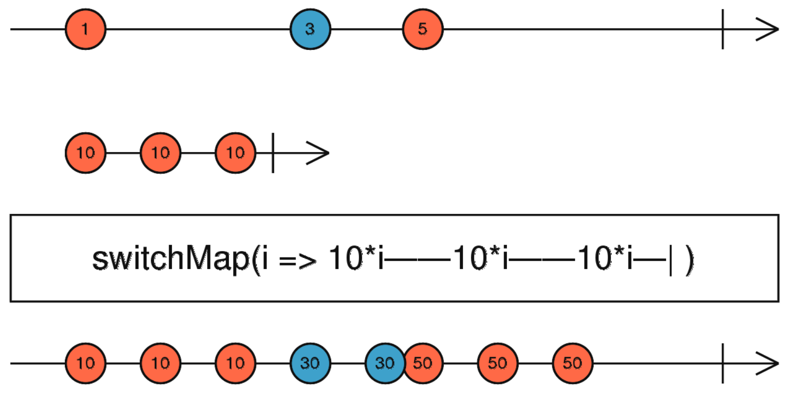
- Observables of Observables
- Inneres Observable «auspacken»

```
const fileObservable = urlObservable.pipe(
  map(url => http.get(url)),
);
```

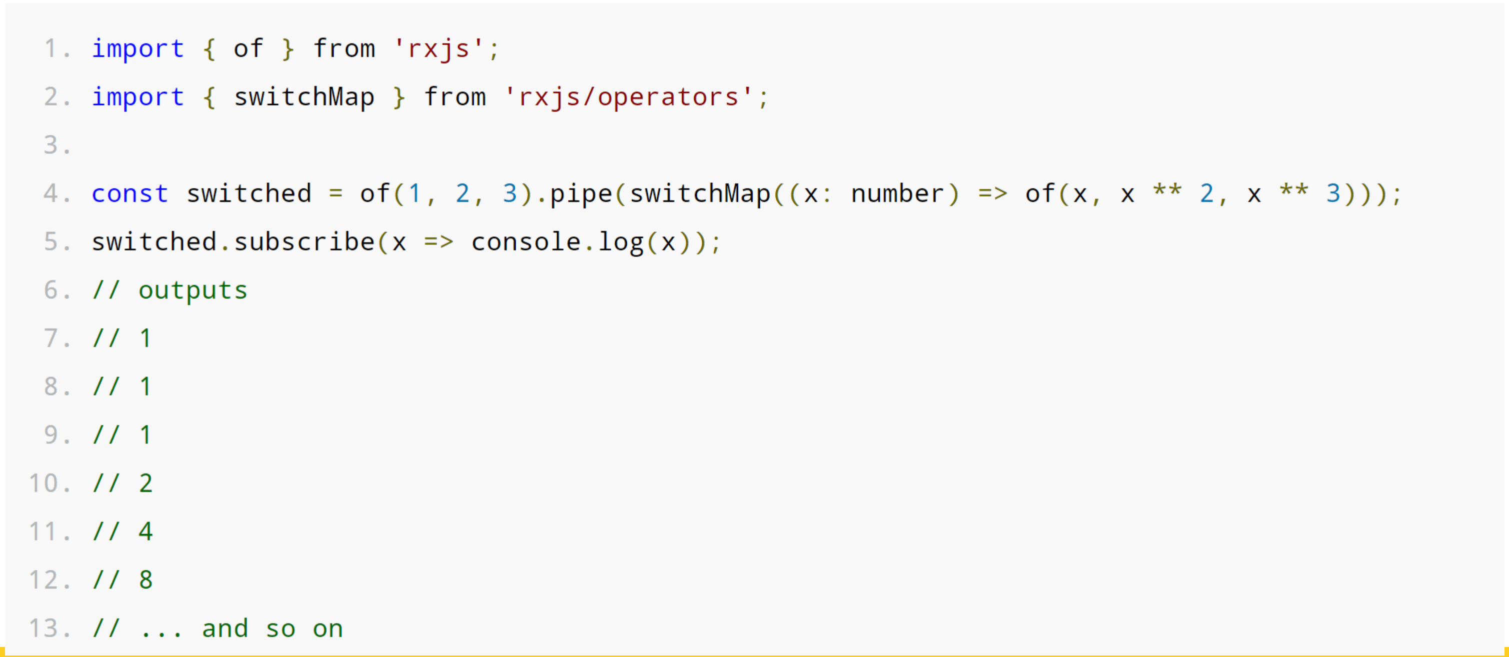
```
const fileObservable = urlObservable.pipe(
  map(url => http.get(url)),
  concatAll(),
);
```

OUTPUT?

switchMap



- Mappt das «innerObservable» nach aussen
- Wenn ein neues Observable ausgegeben wird, werden vorgängige Observables gestoppt nicht weiter ausgeführt
- Demo

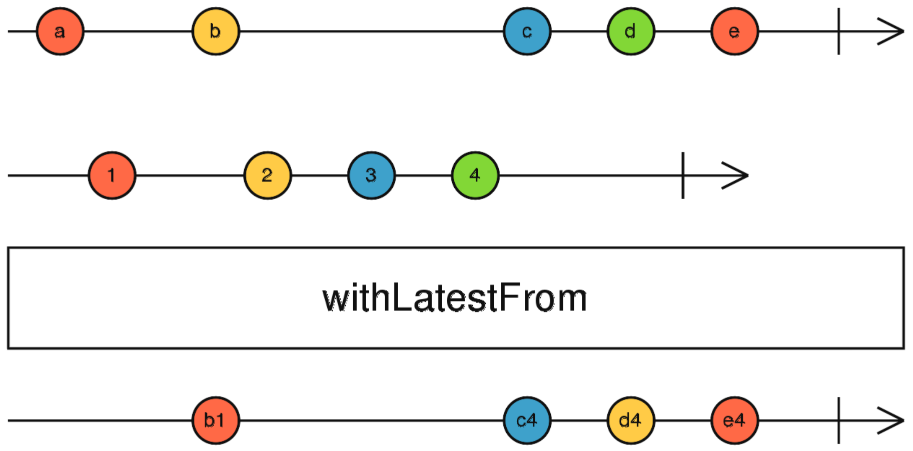


```
import { of } from 'rxjs';
import { switchMap } from 'rxjs/operators';

const switched = of(1, 2, 3).pipe(switchMap((x: number) => of(x, x ** 2, x ** 3)));
switched.subscribe(x => console.log(x));
// outputs
// 1
// 1
// 1
// 2
// 4
// 8
// ... and so on
```

withLatestFrom

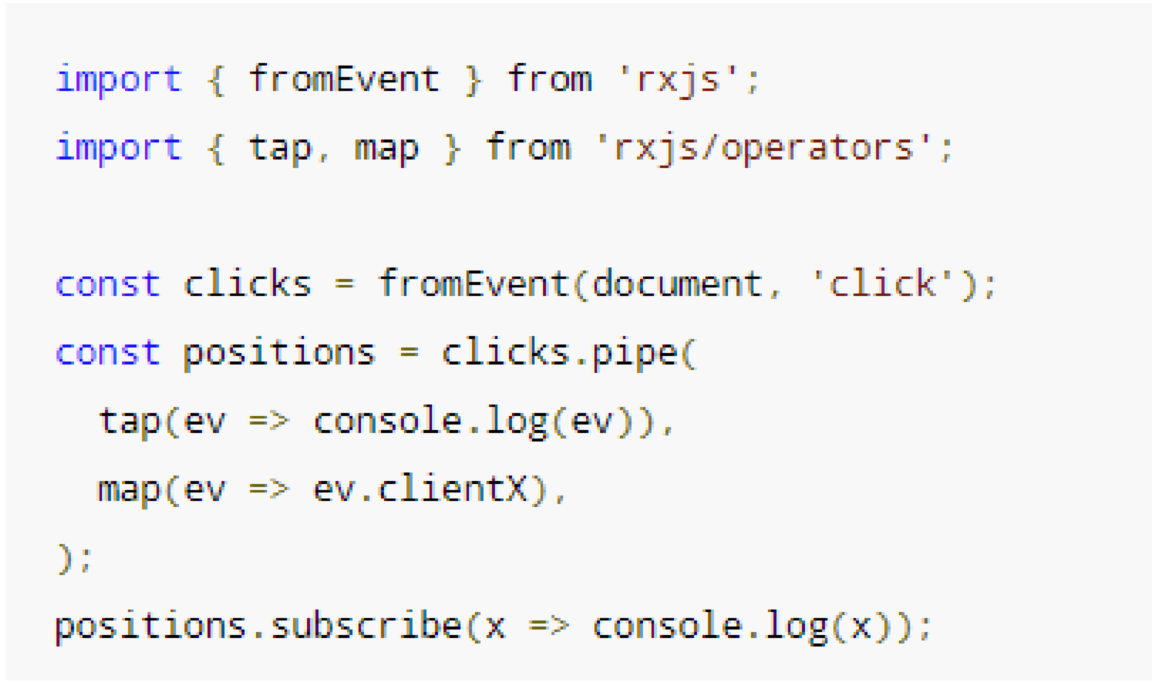
- Kombiniert die Events
- [Demo]()

```
import { fromEvent, interval } from 'rxjs';
import { withLatestFrom } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const timer = interval(1000);
const result = clicks.pipe(withLatestFrom(timer)
result.subscribe(x => console.log(x));
```



tap

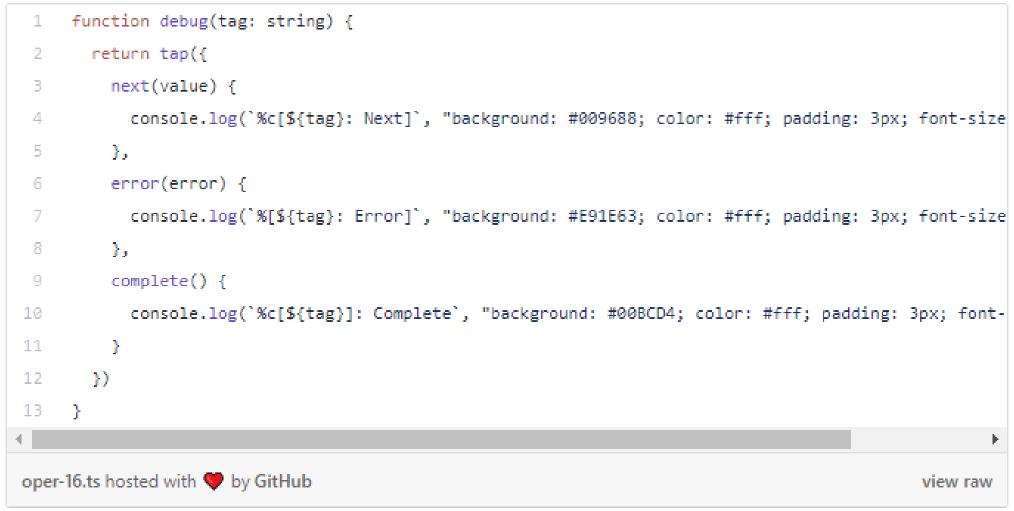
- *Intercepts each emission on the source and runs a function but returns an output which is identical to the source if no errors occur.*
- Demo



```
import { fromEvent } from 'rxjs';
import { tap, map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const positions = clicks.pipe(
  tap(ev => console.log(ev)),
  map(ev => ev.clientX),
);
positions.subscribe(x => console.log(x));
```

Custom Operators

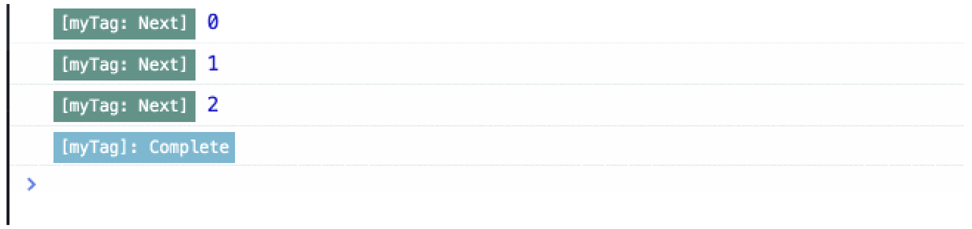
- Zum Beispiel Debug Operator



```
function debug(tag: string) {
  return tap({
    next(value) {
      console.log(`%c[${tag}: Next]`, "background: #009688; color: #fff; padding: 3px; font-size
    },
    error(error) {
      console.log(`%[${tag}: Error]`, "background: #E91E63; color: #fff; padding: 3px; font-size
    },
    complete() {
      console.log(`%c[${tag}]: Complete`, "background: #00BCD4; color: #fff; padding: 3px; font-
    }
  })
}
```

oper-16.ts hosted with ❤ by GitHub view raw

Let's see it in action:



```
[myTag: Next] 0
[myTag: Next] 1
[myTag: Next] 2
[myTag]: Complete
>
```